INF333 - Operating Systems
Lecture VI

Burak Arslan
ext-inf333@burakarslan.com

Galatasaray Üniversitesi

Lecture VI
2024-03-20

Course website

burakarslan.com/inf333

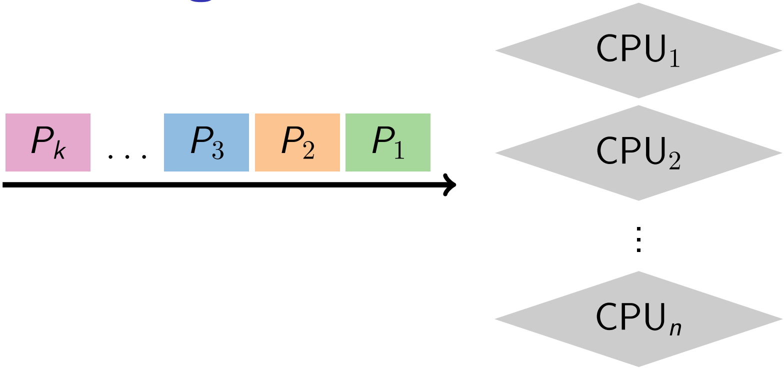
Based On

cs111.stanford.edu

cs212.stanford.edu

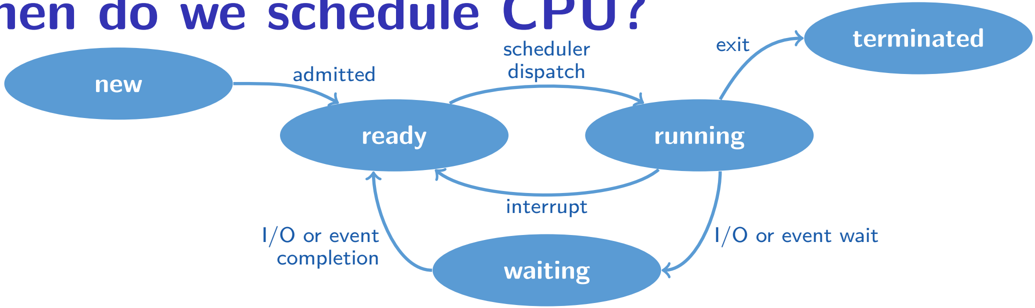
OSC-10 Slides

CPU scheduling



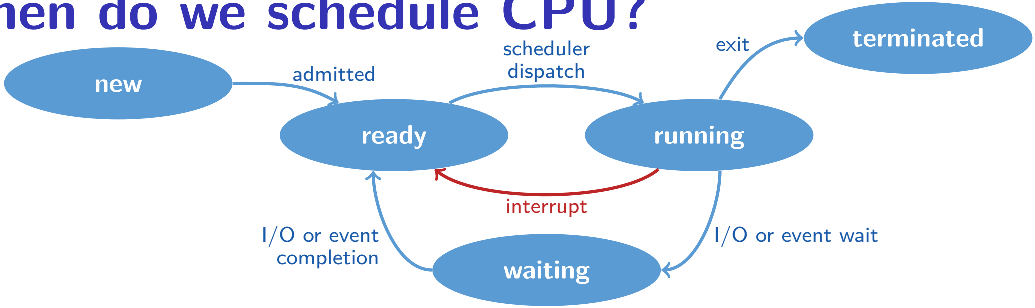
- The scheduling problem:
  - Have k jobs ready to run
  - Have $n \geq 1$ CPUs that can run them
- Which jobs should we assign to which CPU(s)?

When do we schedule CPU?



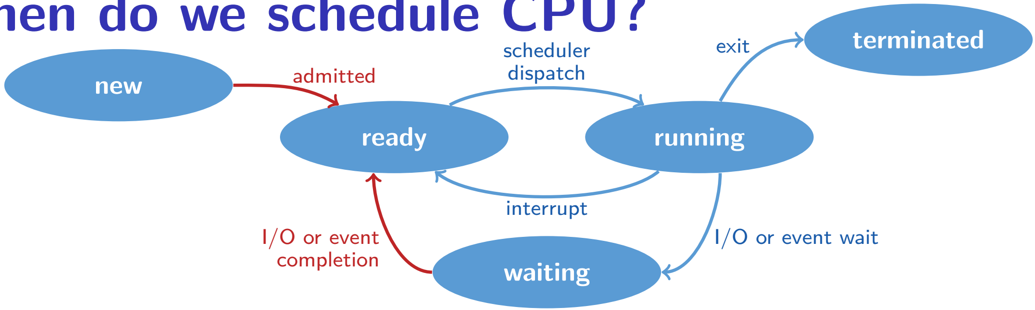
- Scheduling decisions may take place when a process:
    1. Switches from running to ready state
    2. Switches from new/waiting to ready
    3. Switches from running to waiting state
    4. Exits
- Non-preemptive schedules use 3 & 4 only
- Preemptive schedulers run at all four points

When do we schedule CPU?



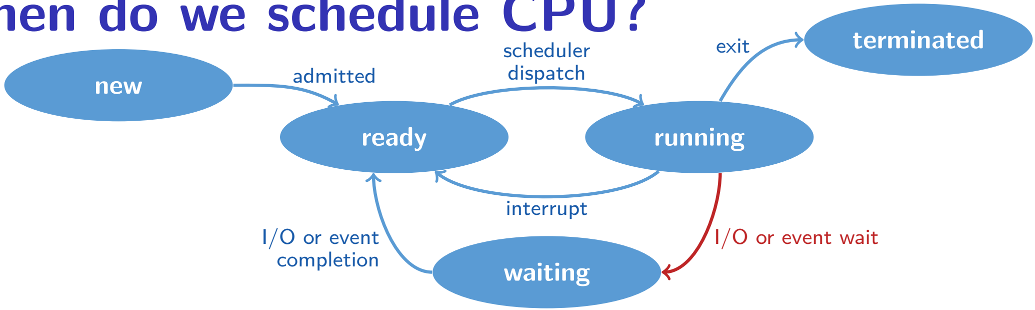
- Scheduling decisions may take place when a process:
  → **1.** Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- Non-preemptive schedules use 3 & 4 only
- Preemptive schedulers run at all four points

When do we schedule CPU?



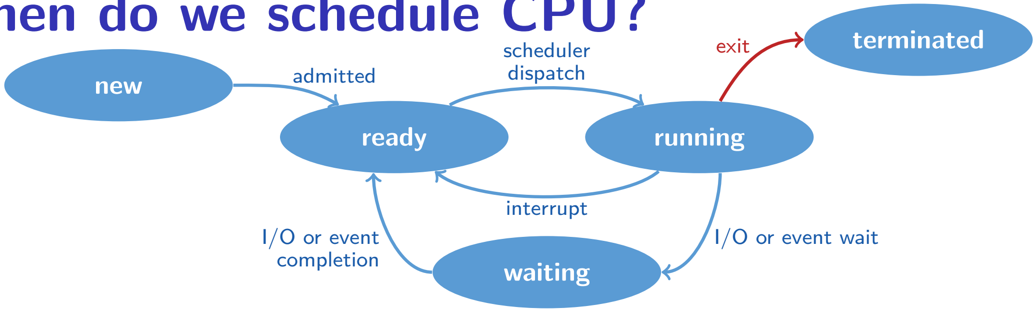
- Scheduling decisions may take place when a process:
    1. Switches from running to ready state
    2. ⟶ Switches from new/waiting to ready
    3. Switches from running to waiting state
    4. Exits
- Non-preemptive schedules use 3 & 4 only
- Preemptive schedulers run at all four points

When do we schedule CPU?



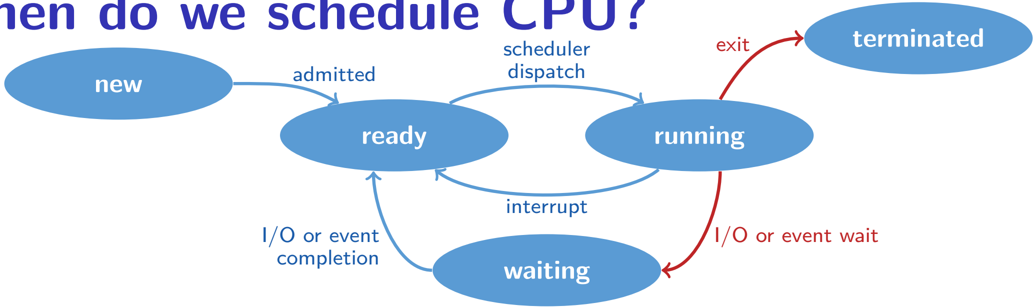
- Scheduling decisions may take place when a process:
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. → Switches from running to waiting state
  4. Exits
- Non-preemptive schedules use 3 & 4 only
- Preemptive schedulers run at all four points

When do we schedule CPU?



- Scheduling decisions may take place when a process:
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- Non-preemptive schedules use 3 & 4 only
- Preemptive schedulers run at all four points

When do we schedule CPU?

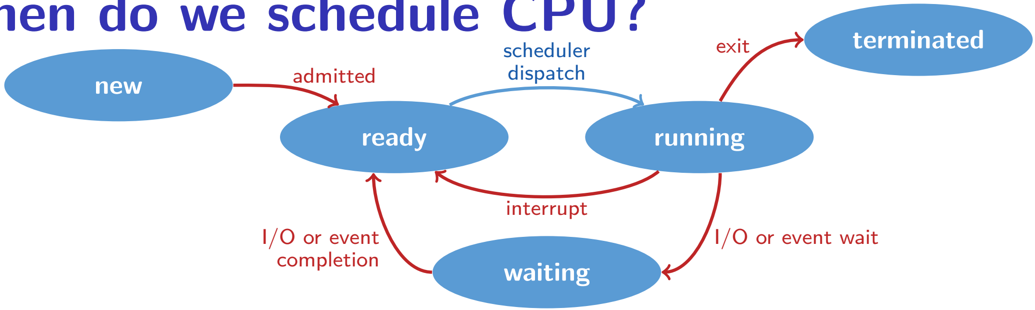


- Scheduling decisions may take place when a process:
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits

⟶ Non-preemptive schedules use 3 & 4 only

- Preemptive schedulers run at all four points

When do we schedule CPU?



- Scheduling decisions may take place when a process:
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- Non-preemptive schedules use 3 & 4 only

⟶ Preemptive schedulers run at all four points

Scheduling criteria

What goals should we have
for a scheduling algorithm?

Scheduling criteria I

Throughput – # of processes that complete per unit time

- Higher is better

Turnaround time – time for each process to complete

- Lower is better

Response time – time from request to first response

- I.e., time between **waiting**→**ready** transition and **ready**→**running** (e.g., key press to echo, not launch to exit)
- Lower is better

Scheduling criteria II

These criteria are affected by secondary criteria

- *CPU utilization* – fraction of time CPU doing productive work
- *Waiting time* – time each process waits in ready queue

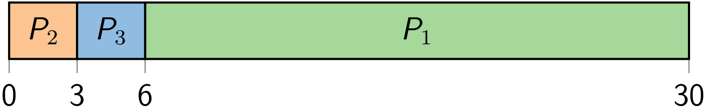
Example: FCFS Scheduling I

- Run jobs in order that they arrive
  - Called "*First-come first-served*" (FCFS)
  - E.g., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2, P_3 arrived immediately after P_1, get:

- Dirt simple to implement—how good is it?
- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround Time: $P_1 : 24$, $P_2 : 27$, $P_3 : 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- Can we do better?

FCFS Scheduling II

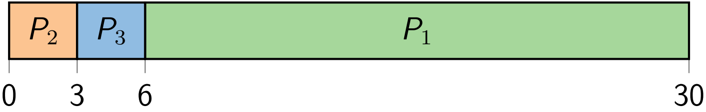
- Suppose we scheduled P_2, P_3, then P_1
  - Would get:



- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround time: $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- Lesson: scheduling algorithm can reduce TT
  - Minimizing waiting time can improve RT and TT
- Can a scheduling algorithm improve throughput?

FCFS Scheduling II

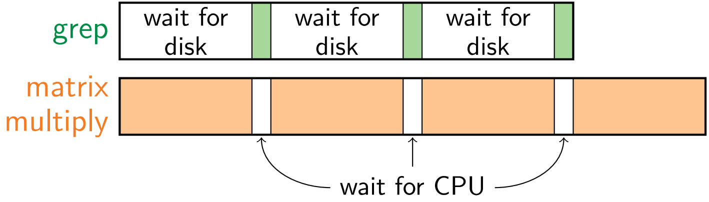
- Suppose we scheduled P_2, P_3, then P_1
  - Would get:



- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround time: $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- Lesson: scheduling algorithm can reduce TT
  - Minimizing waiting time can improve RT and TT
- Can a scheduling algorithm improve throughput?
  - Yes, if jobs require both computation and I/O

View CPU and I/O devices the same

- CPU is one of several devices needed by users' jobs
  - CPU runs compute jobs, Disk drive runs disk jobs, etc.
  - With network, part of job may run on remote CPU
- Scheduling 1-CPU system with n I/O devices like scheduling asymmetric $(n + 1)$-CPU multiprocessor
  - Result: all I/O devices + CPU busy $\Longrightarrow$ $(n + 1)$-fold throughput gain!
- Example: disk-bound grep + CPU-bound matrix multiply
  - Overlap them just right? throughput will be almost doubled



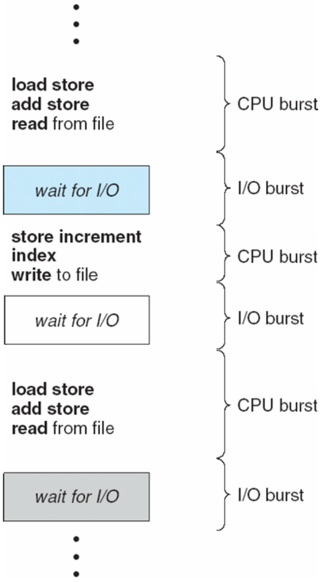
Bursts of computation & I/O

Jobs contain I/O and computation

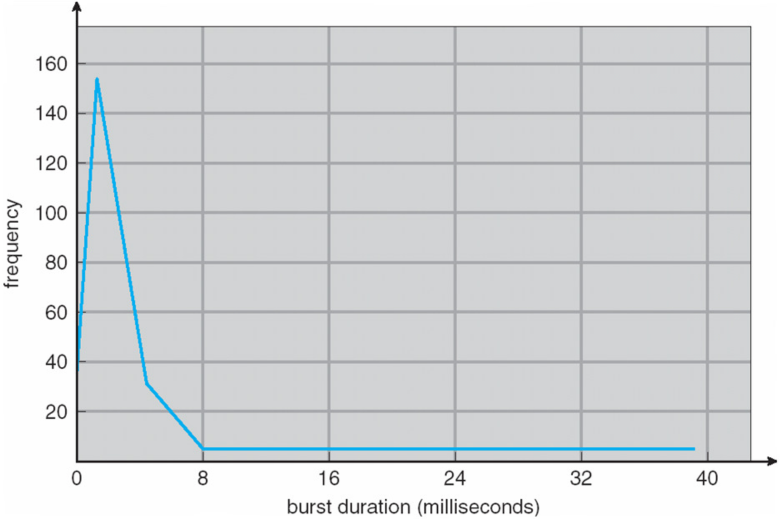
- Bursts of computation
- Then must wait for I/O

To maximize throughput, maximize both CPU and I/O device utilization: How?

- Overlap computation from one job with I/O from other jobs
- Means *response time* very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request



Histogram of CPU-burst times



- What does this mean for FCFS?

FCFS Convoy effect

CPU-bound jobs will hold CPU until exit or I/O
(but I/O rare for CPU-bound thread)

- Long periods where no I/O requests issued, and CPU held
- Result: poor I/O device utilization

FCFS Convoy effect

Example: one CPU-bound job, many I/O bound

- CPU-bound job runs (I/O devices idle)
- Eventually, CPU-bound job blocks
- I/O-bound jobs run, but each quickly blocks on I/O
- CPU-bound job unblocks, runs again
- All I/O requests complete, but CPU-bound job still hogs CPU
- I/O devices sit idle since I/O-bound jobs can't issue next requests

FCFS Convoy effect

Simple hack: run process whose I/O completed

- What is a potential problem?

FCFS Convoy effect

Simple hack: run process whose I/O completed

- What is a potential problem?
  I/O-bound jobs can starve CPU-bound one

SJF Scheduling

- *Shortest-job first* (SJF) attempts to minimize TT
  - Schedule the job whose next CPU burst is the shortest
  - Misnomer unless "job" = one CPU burst with no I/O [term coined for context where there is no I/O, only compute]
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)
- What does SJF optimize?

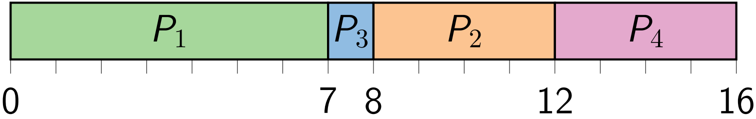
SJF Scheduling

- *Shortest-job first* (SJF) attempts to minimize TT
  - Schedule the job whose next CPU burst is the shortest
  - Misnomer unless "job" = one CPU burst with no I/O [term coined for context where there is no I/O, only compute]
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)
- What does SJF optimize?
  - Gives minimum average *waiting time* for a given set of processes

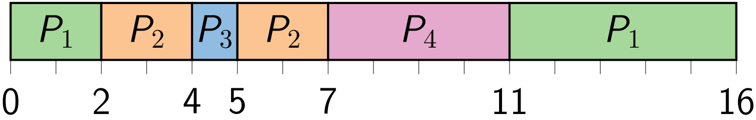
Examples

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Non-preemptive



- Preemptive



SJF limitations

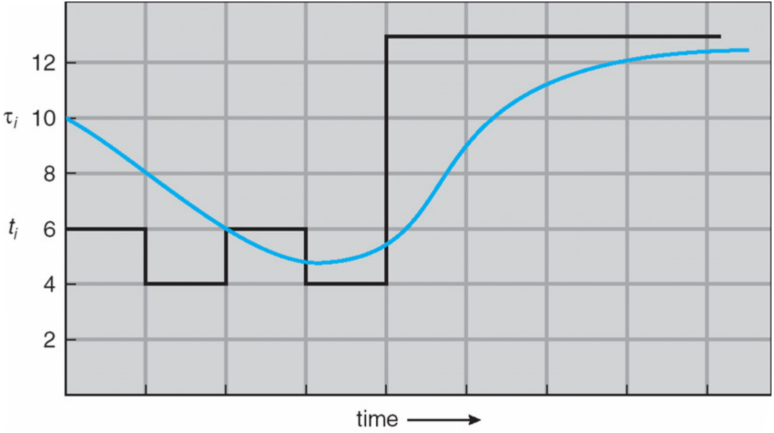
- Doesn't always minimize average TT
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?

- Can lead to unfairness or starvation
- In practice, can't actually predict the future
- But can estimate CPU burst length based on past
  - Exponentially weighted average a good idea
  - t_n actual length of process's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $(n+1)^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

SJF limitations

- Doesn't always minimize average TT
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?
  - Overall longer job has shorter bursts
- Can lead to unfairness or starvation
- In practice, can't actually predict the future
- But can estimate CPU burst length based on past
  - Exponentially weighted average a good idea
  - t_n actual length of process's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $(n+1)^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Exp. weighted average example



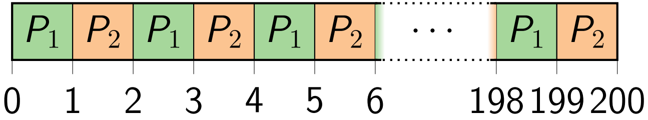
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Round robin (RR) scheduling

- Solution to fairness and starvation
  - Preempt job after some time slice or *quantum*
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)
- Advantages:
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- Disadvantages?

RR disadvantages

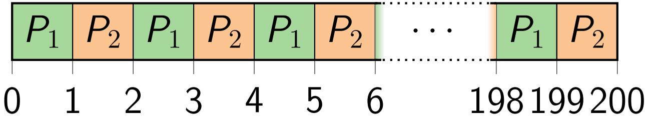
- Varying sized jobs are good ...what about same-sized jobs?
- Assume 2 jobs of time=100 each:



- Even if context switches were free...
  - What would average turnaround time be with RR?
  - How does that compare to FCFS?

RR disadvantages

- Varying sized jobs are good ...what about same-sized jobs?
- Assume 2 jobs of time=100 each:



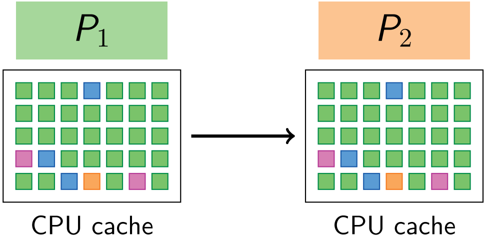
- Even if context switches were free...
  - What would average turnaround time be with RR? *199.5*
  - How does that compare to FCFS? *150*

Context switch costs

- What is the cost of a context switch?

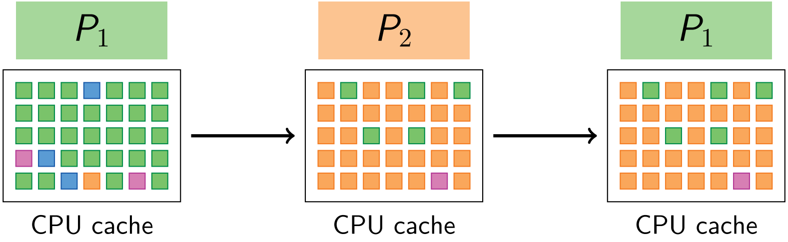
Context switch costs

- What is the cost of a context switch?
- Brute CPU time cost in kernel
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses

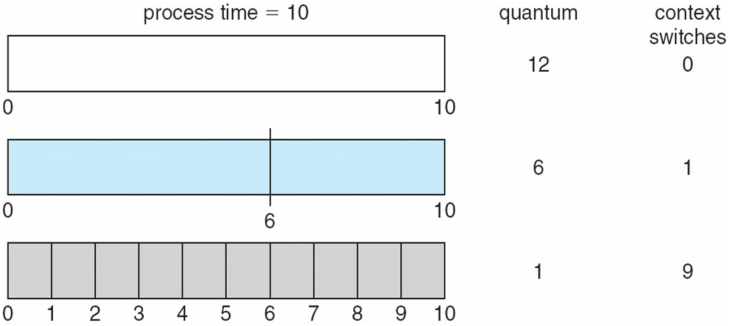


Context switch costs

- What is the cost of a context switch?
- Brute CPU time cost in kernel
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses

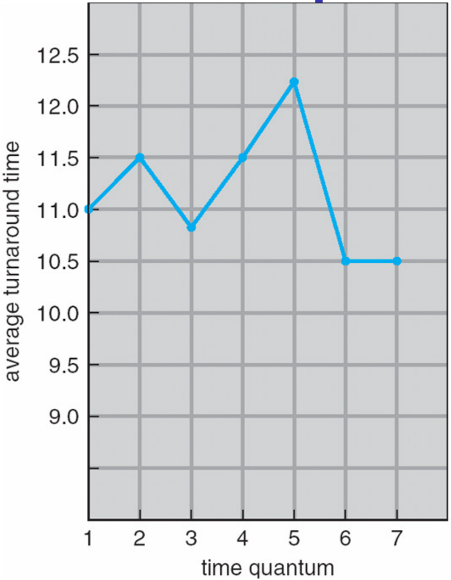


Time quantum



- How to pick quantum?
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS
- Typical values: 1–100 msec

Turnaround time vs. quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Two-level scheduling

- Under memory constraints, may need to *swap* process to disk
- Switching to swapped out process very expensive
  - Swapped out process has most memory pages on disk
  - Will have to fault them all in while running
  - One disk access costs ~10ms. On 1GHz machine, 10ms = 10 million cycles!
- Solution: Context-switch-cost aware scheduling
  - Run in-core subset for "a while"
  - Then swap some between disk and memory
- How to pick subset? How to define "a while"?
  - View as scheduling *memory* before scheduling CPU
  - Swapping in process is cost of memory "context switch"
  - So want "memory quantum" much larger than swapping cost

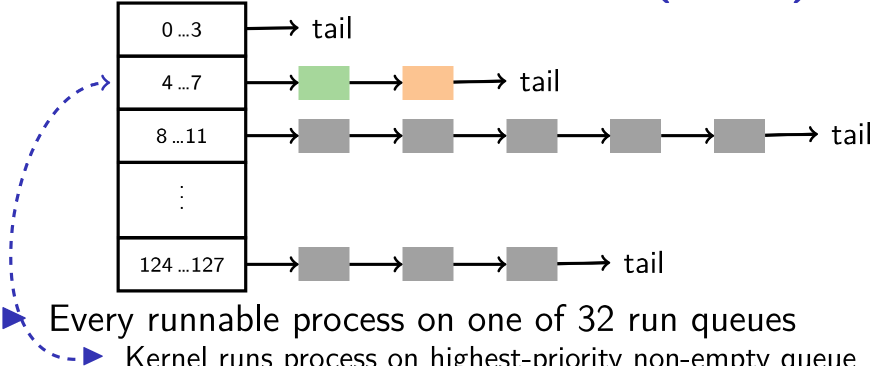
Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?

Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?
  - Aging: increase a process's priority as it waits

Multilevel feeedback queues (BSD)



- Every runnable process on one of 32 run queues
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue
- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- Idea: Favor interactive jobs that use less CPU

Process priority

- p_nice – user-settable weighting factor
- p_estcpu – per-process estimated CPU usage
  - Incremented whenever timer interrupt found process running
  - Decayed every second while process runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right) \texttt{p\_estcpu} + \texttt{p\_nice}$$

  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
- Run queue determined by $\texttt{p\_usrpri}/4$

$$\texttt{p\_usrpri} \leftarrow 50 + \left(\frac{\texttt{p\_estcpu}}{4}\right) + 2 \cdot \texttt{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- p_estcpu not updated while asleep
  - Instead p_slptime keeps count of sleep time
- When process becomes runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \textsf{load}}{2 \cdot \textsf{load} + 1}\right)^{\texttt{p\_slptime}} \times \texttt{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads
- Previous description based on [McKusick] (*The Design and Implementation of the 4.4BSD Operating System*)

Pintos notes

- Same basic idea for second half of project 1
  - But 64 priorities, not 128
  - Higher numbers mean higher priority
  - Okay to have only one run queue if you prefer (less efficient, but we won't deduct points for it)
- Have to negate priority equation:

$$\texttt{priority} = 63 - \left(\frac{\texttt{recent\_cpu}}{4}\right) - 2 \cdot \texttt{nice}$$

Thread scheduling

- With thread library, have two scheduling decisions:
  - *Local Scheduling* – User-level thread library decides which user (green) thread to put onto an available native (i.e., kernel) thread
  - *Global Scheduling* – Kernel decides which native thread to run next
- Can expose to the user
  - E.g., `pthread_attr_setscope` allows two choices
  - `PTHREAD_SCOPE_SYSTEM` – thread scheduled like a process (effectively one native thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
  - `PTHREAD_SCOPE_PROCESS` – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)

Thread dependencies

- Say H at high priority, L at low priority
  - L acquires lock ℓ.
  - Scenario 1 (ℓ a spinlock): H tries to acquire ℓ, fails, spins. L never gets to run.
  - Scenario 2 (ℓ a mutex): H tries to acquire ℓ, fails, blocks. M enters system at medium priority. L never gets to run.
  - Both scenarios are examples of *priority inversion*
- Scheduling = deciding who should make progress
  - A thread's importance should increase with the importance of those that depend on it
  - Naïve priority schemes violate this

Priority donation

- Say higher number = higher priority (like Pintos)
- Example 1: L (prio 2), M (prio 4), H (prio 8)
  - L holds lock ℓ
  - M waits on ℓ, L's priority raised to $L_1 = \max(M, L) = 4$
  - Then H waits on ℓ, L's priority raised to $\max(H, L_1) = 8$
- Example 2: Same L, M, H as above
  - L holds lock ℓ_1, M holds lock ℓ_2
  - M waits on ℓ_1, L's priority now $L_1 = 4$ (as before)
  - Then H waits on ℓ_2. M's priority goes to $M_1 = \max(H, M) = 8$, *and* L's priority raised to $\max(M_1, L_1) = 8$
- Example 3: L (prio 2), $M_1, \ldots M_{1000}$ (all prio 4)
  - L has ℓ, and $M_1, \ldots, M_{1000}$ all block on ℓ. L's priority is $\max(L, M_1, \ldots, M_{1000}) = 4$.

Multiprocessor scheduling issues

Must decide on more than which processes to run

- Must decide on which CPU to run which process

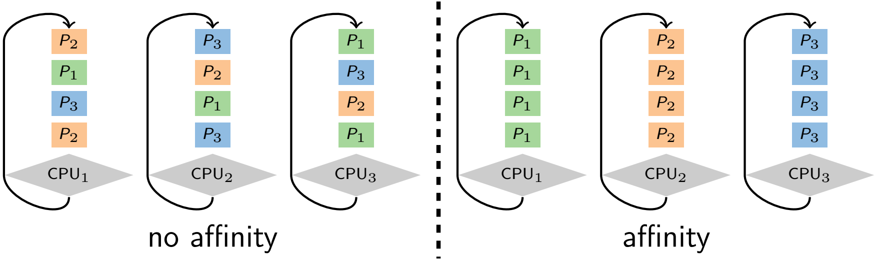
Multiprocessor scheduling issues

Moving between CPUs has costs

- More cache misses, depending on arch. more TLB misses too

Multiprocessor scheduling issues

Affinity scheduling—try to keep process/thread on same CPU



- But also prevent load imbalances
- Do *cost-benefit* analysis when deciding to migrate…
  affinity can also be harmful, when tail latency is critical

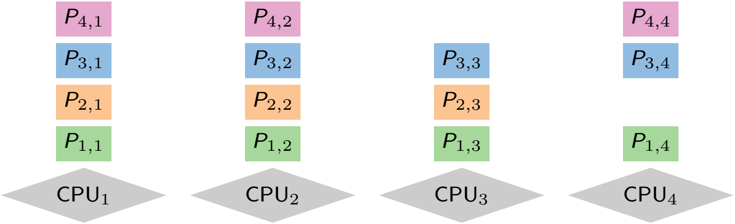
Multiprocessor scheduling

Want related processes/threads scheduled together

- Good if threads access same resources (e.g., cached files)
- Even more important if threads communicate often, otherwise must context switch to communicate

Gang scheduling—schedule all CPUs synchronously

- With synchronized quanta, easier to schedule related processes/threads together



Real-time scheduling

- Two categories:
  - *Soft real time*—miss deadline and audio playback will sound funny
  - *Hard real time*—miss deadline and plane will crash
- System must handle periodic and aperiodic events
  - E.g., processes A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if $\sum \frac{\text{CPU}}{\text{period}} \leq 1$ (not counting switch time)
- Variety of scheduling strategies
  - E.g., first deadline first
    (works if schedulable, otherwise fails spectacularly)

Scheduling with virtual time

- Many modern schedulers employ notion of *virtual time*
  - Idea: Equalize virtual CPU time consumed by different processes
  - Higher-priority processes consume virtual time more slowly
- Forms the basis of the current linux scheduler, CFS
- Case study: Borrowed Virtual Time (BVT) [Duda]
- BVT runs process with lowest *effective virtual time*
  - A_i – *actual virtual time* consumed by process i
  - *effective virtual time* $E_i = A_i - (\text{warp}_i \; ? \; W_i : 0)$
  - Special warp factor allows borrowing against future CPU time …hence name of algorithm

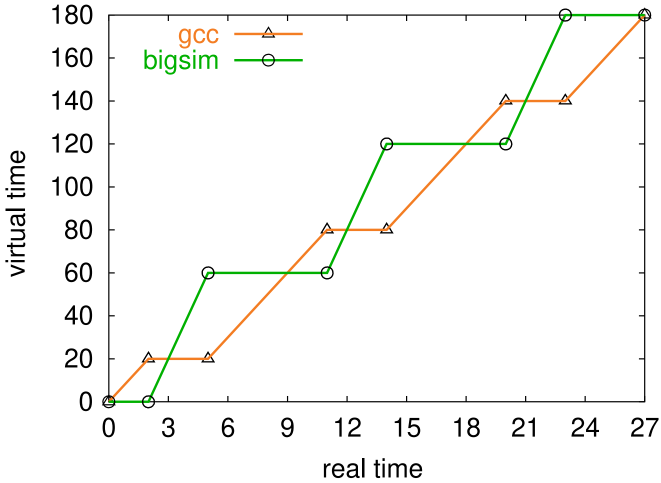
Process weights

- Each process i's faction of CPU determined by weight w_i
  - i should get $w_i / \sum_j w_j$ faction of CPU
  - So w_i is real seconds per virtual second that process i has CPU
- When i consumes t CPU time, track it: A_i += t/w_i
- Example: gcc (weight 2), bigsim (weight 1)
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, ...
  - Lots of context switches, not so good for performance
- Add in context switch allowance, C
  - Only switch from i to j if $E_j \leq E_i - C/w_i$
  - C is wall-clock time ($\gg$ context switch cost), so must divide by w_i
  - Ignore C if j just became runable...why?

Process weights

- Each process i's faction of CPU determined by weight w_i
  - i should get $w_i / \sum_j w_j$ faction of CPU
  - So w_i is real seconds per virtual second that process i has CPU
- When i consumes t CPU time, track it: A_i += t/w_i
- Example: gcc (weight 2), bigsim (weight 1)
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, ...
  - Lots of context switches, not so good for performance
- Add in context switch allowance, C
  - Only switch from i to j if $E_j \leq E_i - C/w_i$
  - C is wall-clock time ($\gg$ context switch cost), so must divide by w_i
  - Ignore C if j just became runable to avoid affecting response time

BVT example



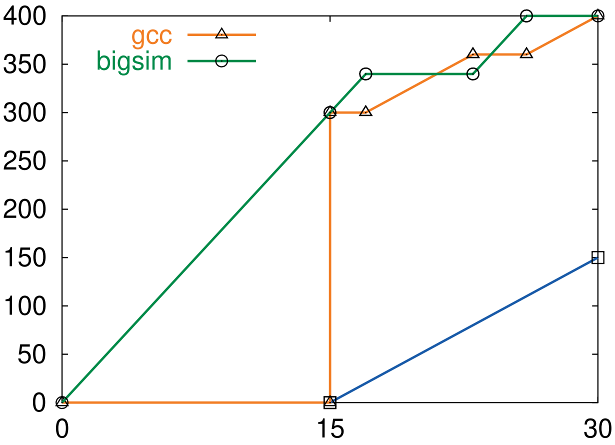
gcc has weight 2, bigsim weight 1, $C = 2$, no I/O

- bigsim consumes virtual time at twice the rate of gcc
- Processes run for C time after lines cross before context switch

Sleep/wakeup

- Must lower priority (increase A_i) after wakeup
  - Otherwise process with very low A_i would starve everyone
- Bound lag with Scheduler Virtual Time (SVT)
  - SVT is minimum A_j for all runnable threads j
  - When waking i from voluntary sleep, set $A_i \leftarrow \max(A_i, SVT)$
- Note voluntary/involuntary sleep distinction
  - E.g., Don't reset A_j to SVT after page fault
  - Faulting thread needs a chance to catch up
  - But do set $A_i \leftarrow \max(A_i, SVT)$ after socket read
- Note: Even with SVT A_i can never decrease
  - After short sleep, might have $A_i >$ SVT, so $\max(A_i, SVT) = A_i$
  - i never gets more than its fair share of CPU in long run

gcc wakes up after I/O



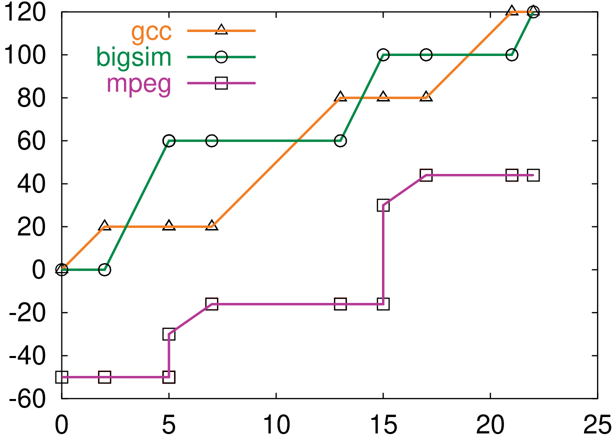
gcc's A_i gets reset to SVT on wakeup

- Otherwise, would be at lower (blue) line and starve bigsim

Real-time threads

- Also want to support time-critical tasks
  - E.g., mpeg player must run every 10 clock ticks
- Recall $E_i = A_i - (\text{warp}_i \text{ ? } W_i : 0)$
  - W_i is *warp factor* – gives thread precedence
  - Just give mpeg player i large W_i factor
  - Will get CPU whenever it is runable
  - But long term CPU share won't exceed $w_i / \sum_j w_j$
- Note W_i only matters when warp_i is **true**
  - Can set warp_i with a syscall, or have it set in signal handler
  - Also gets cleared if i keeps using CPU for L_i time
  - L_i limit gets reset every U_i time
  - $L_i = 0$ means no limit – okay for small W_i value

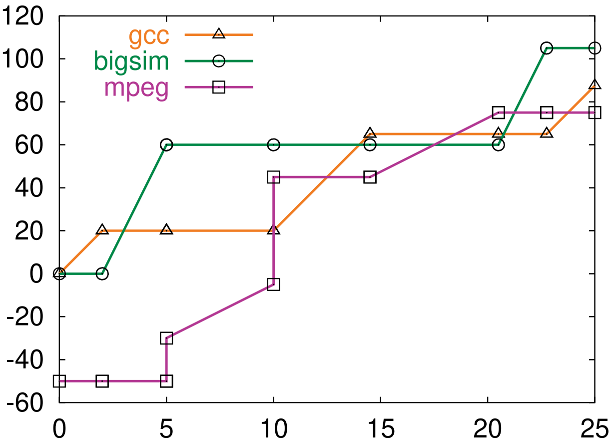
Running warped



mpeg player runs with -50 warp value

- Always gets CPU when needed, never misses a frame

Warped thread hogging CPU



- mpeg goes into tight loop at time 5
- Exceeds L_i at time 10, so warp$_i \leftarrow$ **false**

BVT example: Search engine I

Common queries 150 times faster than uncommon

- Have 10-thread pool of threads to handle requests
- Assign W_i value sufficient to process fast query (say 50)

BVT example: Search engine II

Say 1 slow query, small trickle of fast queries

- Fast queries come in, warped by 50, execute immediately
- Slow query runs in background
- Good for turnaround time

BVT example: Search engine III

Say 1 slow query, but many fast queries

- At first, only fast queries run
- But SVT is bounded by A_i of slow query thread i
- Recall fast query thread j gets $A_j = max(A_j, SVT) = A_j$; eventually $SVT < A_j$ and a bit later $A_j - W_j > A_i$.
- At that point thread i will run again, so no starvation